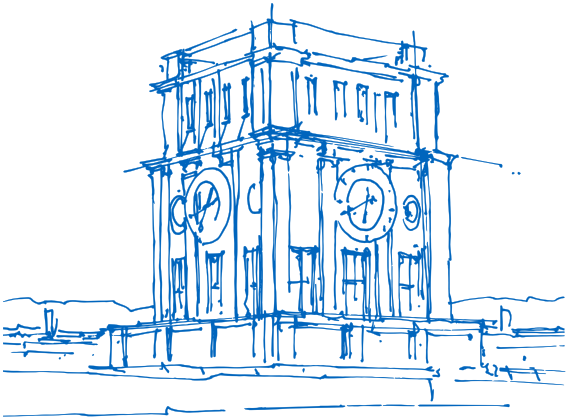
School of Computation, Information and Technology
Technische Universität München

Übung 04: Rekursion & Calling Convention

Einführung in die Rechnerarchitektur



Michael Morandell

School of Computation, Information and Technology
Technische Universität München

11. – 17. November 2024

Mitschriften & Infos

Montags:
https://zulip.in.tum.de/#narrow/stream/2668-ERA-Tutorium—Mo-1000-4

Donnerstags:
https://zulip.in.tum.de/#narrow/stream/2657-ERA-Tutorium—Do-1200-2

Website: https://home.in.tum.de/ momi/era/

Keine Garantie für die Richtigkeit der Tutorfolien.

Bei Unklarheiten/Unstimmigkeiten haben VL/ZÜ-Folien recht!

Inhaltsübersicht

- Wiederholung
- Tutorblatt
  - Calling Convention
  - Rekursion in der Theorie
  - Größter gemeinsamer Teiler

Calling Convention

- “Vertrag“ zwischen allen Entwicklern
- Unterteilung von Register in Caller und Callee saved
  - Caller saved Register dürfen direkt verändert werden
  - Callee saved Register bleiben über Unterprogrammaufrufe erhalten
- https://www.moodle.tum.de/mod/resource/view.php?id=3219780

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Calling Convention

- Datentypen kleiner als 32-Bit werden auf 32-Bit Sign-extended
- 64-Bit Werte in zwei a-Registern pro Wert, aufsteigend sortiert, untere Bit in niedrigerem Register
- >64 Bit werden als Pointer übergeben
- Weitere Parameter werden über den Stack übergeben
- Stack muss immer 16-Byte aligned sein

Rekursion

- Funktion die **sich selbst aufruft**
- Viele Probleme haben rekursive Struktur, dadurch ist Rekursion häufig intuitiv

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times \text{fak}(n-1) & \text{if } n > 0 \end{cases}$$

z.B.
1! = 1
5! = 5 * 4 * 3 * 2 * 1 = 120

```
1 int f(int n) {
2   if (n==0 || n==1)
3     return 1;
4   return n*f(n-1);
5 }
```

(rekrusiv)

```
1 int f(int n) {
2   int a = 1;
3   for(int i=0; i<=n; i++)
4     a = a * i;
5   return a;
6 }
```

(iterativ)

Rekursion: Schema

1. Basisfall (Abbruchbedingung) prüfen
   - Wenn ja –> vordefinierten Wert zurückgeben
   - Wenn nein –> weiter mit rekursiver Berechnung
2. Sicherung von `ra` und evtl. Parametern
3. Vorbereitung der Parameter für den rekursiven Aufruf
4. Rekursiver Aufruf
5. Ergebnis des Aufrufs verwerten
6. Wiederherstellung von `ra`, `sp`
7. Rücksprung

Rekursion: Beispiel

```
1  fun:
2    addi sp, sp, -8
3    sw ra, 0(sp)
4    sw a0, 4(sp)
5    beq a0, zero, end
6    addi a0, a0, -1
7    jal fun
8    end:
9    lw ra, 0(sp)
10   addi sp, sp, 8
11   jalr zero, 0(ra)
```

Achtung: 8 Byte nicht CC-konform, nur zur besseren Darstellung

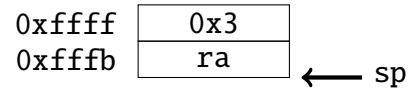
Aufruf mit a0 = 3:

Rekursion: Beispiel

```
1  fun:
2    addi sp, sp, -8
3    sw ra, 0(sp)
4    sw a0, 4(sp)
5    beq a0, zero, end
6    addi a0, a0, -1
7    jal fun
8    end:
9    lw ra, 0(sp)
10   addi sp, sp, 8
11   jalr zero, 0(ra)
```

Achtung: 8 Byte nicht CC-konform, nur zur besseren Darstellung

Aufruf mit a0 = 3:

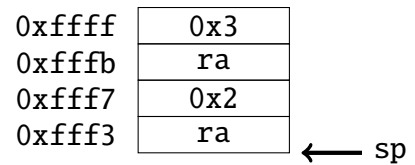


Rekursion: Beispiel

```
1  fun:
2    addi sp, sp, -8
3    sw ra, 0(sp)
4    sw a0, 4(sp)
5    beq a0, zero, end
6    addi a0, a0, -1
7    jal fun
8    end:
9    lw ra, 0(sp)
10   addi sp, sp, 8
11   jalr zero, 0(ra)
```

Achtung: 8 Byte nicht CC-konform, nur zur besseren Darstellung

Aufruf mit a0 = 3:

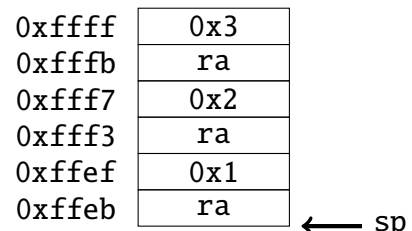


Rekursion: Beispiel

```
1  fun:
2    addi sp, sp, -8
3    sw ra, 0(sp)
4    sw a0, 4(sp)
5    beq a0, zero, end
6    addi a0, a0, -1
7    jal fun
8    end:
9    lw ra, 0(sp)
10   addi sp, sp, 8
11   jalr zero, 0(ra)
```

Achtung: 8 Byte nicht CC-konform, nur zur besseren Darstellung

Aufruf mit a0 = 3:

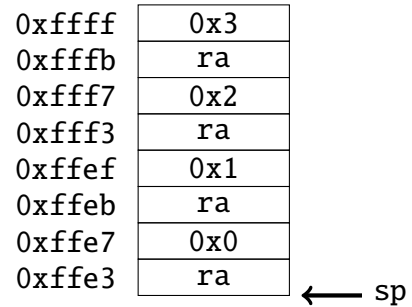


Rekursion: Beispiel

```
1  fun:
2    addi sp, sp, -8
3    sw ra, 0(sp)
4    sw a0, 4(sp)
5    beq a0, zero, end
6    addi a0, a0, -1
7    jal fun
8    end:
9    lw ra, 0(sp)
10   addi sp, sp, 8
11   jalr zero, 0(ra)
```

Achtung: 8 Byte nicht CC-konform, nur zur besseren Darstellung

Aufruf mit a0 = 3:

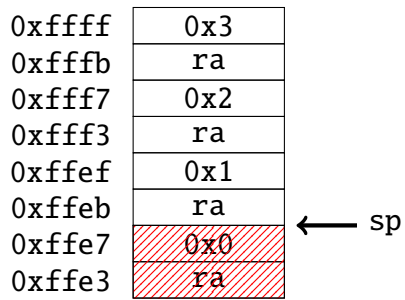


Rekursion: Beispiel

```
1  fun:
2    addi sp, sp, -8
3    sw ra, 0(sp)
4    sw a0, 4(sp)
5    beq a0, zero, end
6    addi a0, a0, -1
7    jal fun
8    end:
9    lw ra, 0(sp)
10   addi sp, sp, 8
11   jalr zero, 0(ra)
```

Achtung: 8 Byte nicht CC-konform, nur zur besseren Darstellung

Aufruf mit a0 = 3:

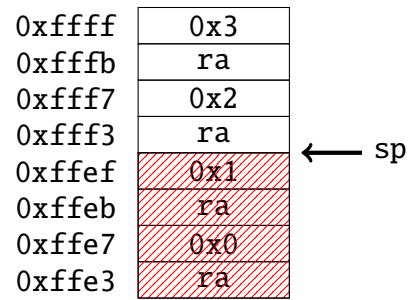


Rekursion: Beispiel

```
1  fun:
2    addi sp, sp, -8
3    sw ra, 0(sp)
4    sw a0, 4(sp)
5    beq a0, zero, end
6    addi a0, a0, -1
7    jal fun
8    end:
9    lw ra, 0(sp)
10   addi sp, sp, 8
11   jalr zero, 0(ra)
```

Achtung: 8 Byte nicht CC-konform, nur zur besseren Darstellung

Aufruf mit a0 = 3:

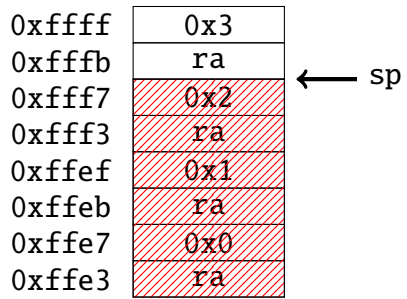


Rekursion: Beispiel

```
1  fun:
2    addi sp, sp, -8
3    sw ra, 0(sp)
4    sw a0, 4(sp)
5    beq a0, zero, end
6    addi a0, a0, -1
7    jal fun
8    end:
9    lw ra, 0(sp)
10   addi sp, sp, 8
11   jalr zero, 0(ra)
```

Achtung: 8 Byte nicht CC-konform, nur zur besseren Darstellung

Aufruf mit a0 = 3:

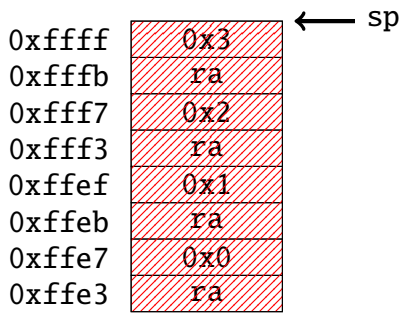


Rekursion: Beispiel

```
1  fun:
2    addi sp, sp, -8
3    sw ra, 0(sp)
4    sw a0, 4(sp)
5    beq a0, zero, end
6    addi a0, a0, -1
7    jal fun
8    end:
9    lw ra, 0(sp)
10   addi sp, sp, 8
11   jalr zero, 0(ra)
```

Achtung: 8 Byte nicht CC-konform, nur zur besseren Darstellung

Aufruf mit a0 = 3:



Endrekursion (Tail-Recursion)

- Rekursiver Aufruf ist der letzte Schritt in der Funktion

- **Warum ist Endrekursion effizient?**
  - Rückkehradresse muss nicht gespeichert werden
  - Speicherverbrauch wird stark reduziert, insbesondere bei tiefen rekursiven Aufrufen.
- **Verwendung der ‘tail‘-Pseudoinstruktion in RISC-V**
  - ‘tail <funktion>‘ ersetzt normalen Funktionsaufruf durch optimierten rekursiven Sprung
  - Intern wird die ‘tail‘-Pseudoinstruktion in zwei Instruktionen umgewandelt:
    1. ‘auipc x6, offset[31:12]‘ – Lädt den oberen Teil des Offsets in das Register x6 (t1).
    2. ‘jalr x0, x6, offset[11:0]‘ – Springt zum Ziel und gibt die Kontrolle nicht zurück.

Fragen?

Bis zum nächsten Mal ;)

Folien inspiriert von Niklas Ladurner